CRC-8 firmware implementations for SMBus

John Milios
USAR Systems
Scope

- CRC-8 has been defined as optional feature of SMBus V1.1
- Packet Error Checking will improve the reliability of the bus
- Let’s make sure that we understand it the same way
CRC’s basic idea

- The remainder of a division changes a lot with small changes in the dividend

```
1001
Divisor
```

```
0000001101 = Quotient
0001100001 = Dividend
0001,. ., 0000,. ., 0011,. ., 0000,. ., 0110,. ., 0000, ., 1100,. , 1001,. , 1010,. , 1001,. , 0110, 0000, 1100 1001 101
```
Polynomial arithmetic mod 2

- No propagation of carry
- Addition AND subtraction are performed with XOR
- Multiplication is similar with binary except that sums are calculated with XORs
- Division relies on “weak” definition of larger or equal
  - X is greater than or equal to Y if and only if the position of the highest 1 bit of X is the same or greater of the highest 1 bit of Y
CRC calculation

• We need the following:
  • A polynomial of width M (divisor). In the case of SMBus, the polynomial will be using is $X^8 + X^2 + X + 1$. The width of this polynomial is 8 (the highest power of $X$ indicates the width) and it can be represented as 1 0000 0111. Since the width of the polynomial is 8 we refer to our CRC method as CRC-8.
  • A message represented as a bit-stream augmented with $M = 8$ zeroes at the end.
  • Division of the augmented bit-stream message by the polynomial 0000 0111. The remainder will be the CRC-8 check byte.
CRC calculation example

- Our polynomial is: 10000 0111.
  The original message is: 0101 1100.
  After we augment it with 8 zeroes it becomes: 0101 1100 0000 0000
  We ignore the quotient as it is not used in the CRC calculation.

0101110000000000 =CRC
0111111111100000 =CRC
010110001000 =CRC
010110001000 =CRC
010110001000 =CRC
010110001000 =CRC
What to do with CRC

- The original message can be appended with the calculated CRC-8 byte and transmitted to the receiver.
  - The CRC-8 enhanced message will be 0101 1100 10010011.

- The receiver can perform the same calculation upon the first 8 bits of the original message and then compare the result with the received CRC-8.

- Or calculate the CRC-8 upon the whole (16 bits in this example) message, without appending zeroes and verify that the outcome is zero.
The algorithm for the example

- Initialize the shift register with zeroes
- Shift left the augmented message until a 1 comes out
- XOR the contents of the register with the low eight bits of the polynomial
- Continue until the whole message has passed through the shift register (the trailing zero bits will be the last ones to fill the register and their role is to push out the original message)
- The CRC-8 is the last value of the shift register

Calculator Example
Implementation problems

- Easier to implement in hardware
- In real communication systems including SMBus you may not want to put the calculation load at the end
  - It takes long time
  - You need to respond fast with ACK, NACK
- How can you perform the calculation byte by byte?
Back to arithmetic...

• What we need is $A/B = Q + R$
• In CRC we are interested in only on the remainder so we can re-write our expression as $A/B => R$
• If we express $A = A_1 + A_2$ then our task would be described as
  • $(A_1 + A_2)/B => R$
  • or $A_1/B + A_2/B = (Q_1 + R_1/B) + A_2/B$
  • or $Q_1 + (R_1 + A_2)/B => R$
Two different methods

Straightforward

Breaking the message into a sum

\[
\begin{align*}
0101 & 10000000 = \text{CRC} \\
11010 & \text{XOR with polynomial in 2}\text{nd position} \\
0110000000 = \text{CRC} \\
11010 & \text{XOR with polynomial in 3}\text{rd position} \\
0001000000 = \text{CRC} \\
11010 & \text{XOR with polynomial in 6}\text{th position} \\
0101000 = \text{CRC} \\
11010 & \text{XOR with polynomial in 7}\text{th position} \\
011100 = \text{CRC} \\
11010 & \text{XOR with polynomial in 8}\text{th position} \\
00110 = \text{CRC}
\end{align*}
\]

\[
\begin{align*}
0101 & \text{CRC} \\
11010 & \text{XOR with polynomial (2}\text{nd position)} \\
0111000 = \text{CRC} \\
11010 & \text{XOR with polynomial (3}\text{rd position)} \\
001100 = \text{CRC (completed calculation with first number)} \\
10000000 & \text{Add (XOR) now the remainder to the second nibble} \\
01000000 = \text{CRC} \\
11010 & \text{XOR with polynomial (6}\text{th position)} \\
0101000 = \text{CRC} \\
11010 & \text{XOR with polynomial (7}\text{th position)} \\
011100 = \text{CRC} \\
11010 & \text{XOR with polynomial (8}\text{th position)} \\
00110 = \text{CRC}
\end{align*}
\]
Interesting observations

- We can use the CRC value derived from the first nibble and add it (XOR) to the second nibble in order to continue the calculation.
- The XORs of the dividend with the polynomial occur in the same bit positions.
- If we XOR the polynomial shifted into these positions we will derive the original bit-stream augmented with the CRC as it is shown below:

```
_11010
__11010
_____11010
______11010
________11010
010110000110
```
1st method - Direct

- Initialize the CRC register with zeroes
- Perform the straightforward algorithm (bit by bit) on the first byte augmented by 8 zeroes
- XOR the CRC with the next byte and calculate again with the straightforward method
- Until the whole message is passed through the calculation
**Direct method sample code**

* This code calculates a new CRC value for each byte received
* Registers used:
  * Accumulator A
  * Index X
  * Memory register CRC
  * Memory register temp
* Input: Memory register CRC contains the previously calculated CRC value - initialized to zero
  * Accumulator A contains the next input data byte
* Output: Memory register CRC contains the current CRC value
* Minimum overhead to enter: [6] (jump to subroutine call)
* [x] denotes machine cycles for each instruction

```plaintext
init_crc:
  ldx #8 [2]; initialize x-register with the # of bits to be shifted
  eor CRC [4]; X-OR new byte with contents of memory location CRC in order to obtain
  (remainder + next incoming byte)
  crc_loop:
  * start the straightforward approach
    rola [3]; rotate left and place the MSB into the Carry
  * rola operation fills the empty bits with zeroes
    bcc zero [3]; if carry is clear no need to do anything, continue
    eor #$07 [2]; else, perform the X-OR of CRC with the polynomial
  zero:
    decx [3]; decrement bit counter
    bne crc_loop [3]; if more bits need to be processed repeat the loop
    sta CRC [5]; save the new CRC value
  return:
    rts [6]; no more bits, return from subroutine
```
## Direct method performance

<table>
<thead>
<tr>
<th>Fosc</th>
<th>Execution time for one byte</th>
<th>Overhead for “5 bytes” protocols</th>
<th>Overhead for “35 bytes” protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Msg Transfer Time*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 1.5 ms</td>
<td></td>
</tr>
<tr>
<td>4 MHz</td>
<td>68us</td>
<td>340 us</td>
<td>2.36 ms</td>
</tr>
<tr>
<td>2 MHz</td>
<td>135 us</td>
<td>680 ms</td>
<td>4.75 ms</td>
</tr>
<tr>
<td>1 MHz</td>
<td>270 us</td>
<td>1.35 ms</td>
<td>9.45 ms</td>
</tr>
<tr>
<td>600 KHz</td>
<td>450 us</td>
<td>2.25 ms</td>
<td>15.75 sec</td>
</tr>
</tbody>
</table>

Machine cycles required = 135
SM Bus operates at 30KHz
Recall

- All it matters is the positions of the polynomial when shifted
- We know the positions if we know the data byte
- We can ADD (XOR) the CRC of the first byte with the next incoming byte
- All we need is a table
- Calculator example
2nd method- 256 lookup table

- Initialize the CRC register to 0
- XOR each incoming byte with the previous CRC value (Add the remainder of the previous division with the new value). The result is the new byte that we need to calculate the CRC value (or remainder of the division)
- Use this value as the index to the table to obtain the new remainder
- Continue until you have passed all bytes through the process
- The last byte retrieved from the table is the final CRC value
Code example

* This code performs a table lookup to determine the new CRC value for each byte received
* Registers used:
  * Accumulator A
  * Memory register CRC
* Input: Memory register CRC contains the previously calculated CRC value
  * Accumulator A contains the input data byte
* Output: Memory register CRC contains the current CRC value

get_crc:
  eor CRC  [4] ; X-OR the received byte with the stored CRC value (result in ACC)
  tax [2] ; no need to pay taxes yet... just transfer ACC to X register
  lda table,x [5] ; load the new CRC value from the lookup table
  sta CRC [4] ; save the new CRC value

Calculator example
2nd method performance

<table>
<thead>
<tr>
<th>Fosc</th>
<th>Execution time for one byte</th>
<th>Overhead for “5 bytes” protocols</th>
<th>Overhead for “35 bytes” protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Msg Transfer Time* = 1.5 ms</td>
<td>Msg transfer time = 10.5 ms</td>
</tr>
<tr>
<td>4 MHz</td>
<td>7.5 us</td>
<td>37.5 us</td>
<td>262 us</td>
</tr>
<tr>
<td>2 MHz</td>
<td>15 us</td>
<td>75 us</td>
<td>525 us</td>
</tr>
<tr>
<td>1 MHz</td>
<td>30 us</td>
<td>150 us</td>
<td>1 ms</td>
</tr>
<tr>
<td>600 KHz</td>
<td>50 us</td>
<td>250 us</td>
<td>1.75 ms</td>
</tr>
</tbody>
</table>

Machine cycles required = 15
Working with nibbles

• The same method can be applied to nibbles instead of bytes
• Results in a much smaller table (16 bytes)
• Good for 4-bit MCUs
• A lot of shifts for 8-bit MCUs
• But it saves ROM space and it is faster than the direct method calculation
  • Calculator example
Performance

<table>
<thead>
<tr>
<th>Fosc</th>
<th>Execution time for one byte (us)</th>
<th>Overhead for “5 bytes” protocols</th>
<th>Overhead for “35 bytes” protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Msg Transfer Time*</td>
<td>Msg transfer time =</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>10.5 ms</td>
</tr>
<tr>
<td>4 MHz</td>
<td>45</td>
<td>225 us</td>
<td>1575 us</td>
</tr>
<tr>
<td>2 MHz</td>
<td>90</td>
<td>450 us</td>
<td>3150 us</td>
</tr>
<tr>
<td>1 MHz</td>
<td>180</td>
<td>900 us</td>
<td>6300 us</td>
</tr>
<tr>
<td>600 KHz</td>
<td>300</td>
<td>1500 us</td>
<td>10500 us</td>
</tr>
</tbody>
</table>

Machine cycles required = 90
Improved 32 byte table lookup

• We can eliminate some of the shifts by creating a second 16 byte table with reversed nibbles
• Example
  • 07   ->  70
• Improves performance in 8-bit MCUs
# 32 byte table performance

<table>
<thead>
<tr>
<th>Fosc</th>
<th>Execution time for one byte (us)</th>
<th>Overhead for “5 bytes” protocols</th>
<th>Overhead for “35 bytes” protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Msg Transfer Time*</td>
<td>Msg transfer time =</td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 1.5 ms</td>
<td>10.5 ms</td>
</tr>
<tr>
<td>4 MHz</td>
<td>30</td>
<td>150 us</td>
<td>1050 us</td>
</tr>
<tr>
<td>2 MHz</td>
<td>60</td>
<td>300 us</td>
<td>2100 us</td>
</tr>
<tr>
<td>1 MHz</td>
<td>120</td>
<td>600 us</td>
<td>4200 us</td>
</tr>
<tr>
<td>600 KHz</td>
<td>199</td>
<td>995 us</td>
<td>6965 us</td>
</tr>
</tbody>
</table>

Machine cycles required = 60
CRC-8 polynomial

- Polynomial: $X^8 + X^2 + X^1 + X^0$
- It will detect all single bit errors
- All Odd number of bit errors
- Any burst error less than 8 bits long
- Most of other type of errors
Conclusions

• By understanding the CRC principles algorithms can be devised to address any speed/space condition
• It is important for the CRC-8 results to agree
• The CRC calculator will be made available along with the source code at www.usar.com
• CRC-8 calculator and white paper available for download from USAR’s web-site